CS1010S Programming Methodology

Lecture 8
Implementing Data Structures

18 March 2015

Saying of the Wise

I hear and I forget.
I see and I remember.
I do and I understand.

- Xun Zi

不闻不若闻之
闻之不若见之
见之不若知之
知之不若行之
学至于行之而止矣

Some Philosophy

Python is like chess

- Easy to learn, hard to master

Levels of learning

- Knowledge
- Understanding
- Application

Lecture

Trainings

Recitation

Tutorial

Missions

Secret Recipe for Success in CS1010S

- DOING IT (Getting hands dirty and writing code)!

Lots of Optional Trainings

Python you should know

Python Statements :

- `def`
- `return`
- `lambda`
- `if, elif, else`
- `for, while, break, continue`
- `import`

Data abstraction primitives:

- `tuple`
- `list`

Re-Midterm

20 March, 7 pm

SoC SR1

Today's Agenda

- The Game of Nim
  - More Wishful Thinking
  - Understanding Scheme code
  - Simple data structures
- Designing Data Structures
- Multiple Representations

The Game of Nim

- Two players
- Game board consists of piles of coins
- Players take turns removing any number of coins from a single pile
- Player who takes the last coin wins

Let’s Play!!

How to Write This Game?

1. Keep track of the game state
2. Specify game rules
3. Figure out strategy
4. Glue them all together

Let’s start with a simple game with two piles

Start with Game State

What do we need to keep track of?

Number of coins in each pile!

Game State

Wishful thinking:

Assume we have:

```
def make_game_state(n, m):
        ...
```

where `n` and `m` are the number of coins in each pile.

What Else Do We Need?

```
def size_of_pile(game_state, p):
    ...
```

where p is the number of the pile

```
def remove_coins_from_pile(game_state,
                           n, p):
    ...
```

where p is the number of the pile and n is the number of coins to remove from pile p.

Let’s start with the game

```
def play(game_state, player):
    display_game_state(game_state)
    if is_game_over(game_state):
        announce_winner(player)
    elif player == "human":
        play(human_move(game_state), "computer")
    elif player == "computer":
        play(computer_move(game_state), "human")
```

What happens if we evaluate:

```
play(make_game_state(5, 8), "mickey-mouse")
```

Take Care of Error Condition

```
def play(game_state, player):
    display_game_state(game_state)
    if is_game_over(game_state):
        announce_winner(player)
    elif player == "human":
        play(human_move(game_state), "computer")
    elif player == "computer":
        play(computer_move(game_state), "human")
    else:
        print("player wasn't human or computer:",
              player)
```

Displaying Game State

```
def display_game_state(game_state):
    print("")
    print(" Pile 1: " +
          str(size_of_pile(game_state,1)))
    print(" Pile 2: " +
          str(size_of_pile(game_state,2)))
    print("")
```

Game Over

Checking for game over:

```
def is_game_over(game_state):
    return total_size(game_state) == 0

def total_size(game_state):
    return size_of_pile(game_state, 1) +
           size_of_pile(game_state, 2)
```

Announcing winner/loser:

```
def announce_winner(player):
    if player == "human":
        print("You lose. Better luck next time.")
    else:
        print("You win. Congratulations.")
```

Getting Human Player’s Move

```
def human_move(game_state):
    p = prompt("Which pile will you remove from?")
    n = prompt("How many coins do you want to remove?")
    return remove_coins_from_pile(game_state,
                                  int(n),
                                  int(p))


def prompt(prompt_string):
    return input(prompt_string)
```

Artificial Intelligence

```
def computer_move(game_state):
    pile = 1 if size_of_pile(game_state, 1) > 0 else 2
    print("Computer removes 1 coin from pile "
          + str(pile))
    return remove_coins_from_pile(game_state, 1, pile)
```

Is this a good strategy?

Game State

```
def make_game_state(n, m):
    return (10 * n) + m

def size_of_pile(game_state, pile_number):
    if pile_number == 1:
        return game_state // 10
    else:
        return game_state % 10

def remove_coins_from_pile(game_state,
                           num_coins, pile_number):
    if pile_number == 1:
        return game_state – 10 * num_coins
    else:
        return game_state - num_coins
```

What is the limitation of this representation?

Another Implementation

```
def make_game_state(n, m):
    return (n, m)

def size_of_pile(game_state, p):
    return game_state[p-1]
```

Another Implementation

```
def remove_coins_from_pile(game_state,
                           num_coins,
                           pile_number):
  if pile_number == 1:
    return make_game_state(size_of_pile(game_state,1)
           - num_coins, size_of_pile(game_state,2))
  else:
    return make_game_state(size_of_pile(game_state,1),
                           size_of_pile(game_state,2)
                           - num_coins)
```

Improving Nim

Lets modify our Nim game by allowing “undo”

- Only Human player can undo, not Computer
- Removes effect of the most recent move
  - i.e. undo most recent computer and human move
  - Human’s turn again after undo
- Human enters “0” to indicate undo

Key Insight

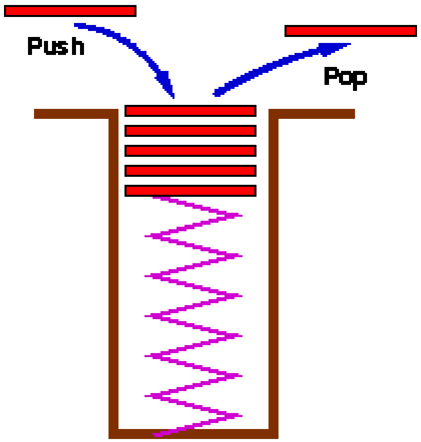
We need a data structure to remember the history of game states

History of game states

- Before each human move, add the current game state to the history.
- When undoing,
  - Remove most recent game state from history
  - Make this the current game state

Data structure: Stack

- A stack is a data structure with the LIFO property.
  - Last In, First Out
  - Items are removed in the reverse order in which they were added.



Wishful thinking again

Assume we have the following:

`make_stack()` : returns a new, empty stack

`push(s, item)` : adds item to stack s

`pop(s)` : removes the most recently added item from stack s, and returns it

`is_empty(s)` : returns `True` if s is empty, `False` otherwise

Stack operations

```
>>> s = make-stack()
>>> pop(s)
None
```
empty stack, nothing to pop

```
>>> push(s, 5)
>>> push(s, 3)
>>> pop(s)
3
>>> pop(s)
5
is_empty(s)
True
```

Implement a stack as homework

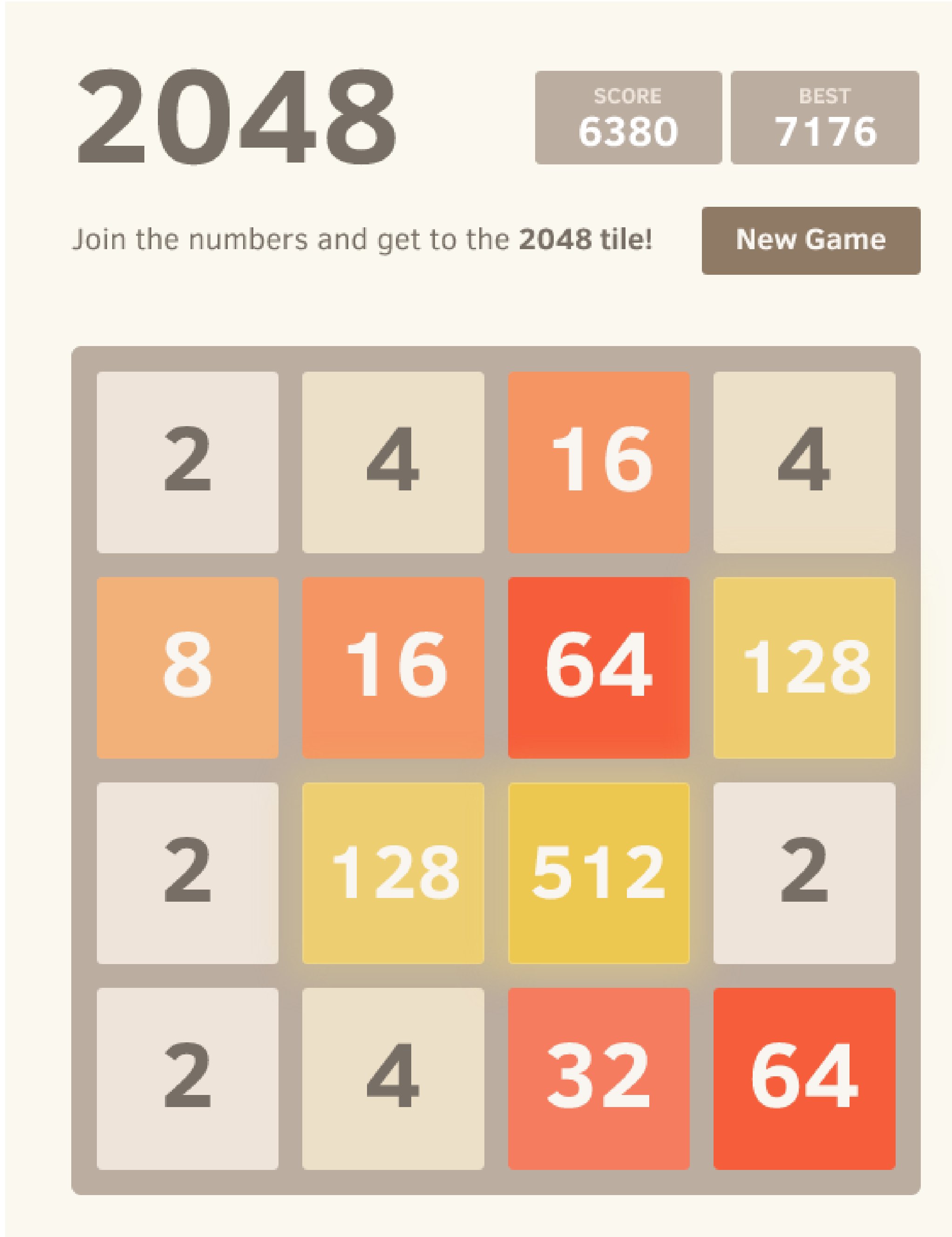
Changes to Nim

```
game_stack = make_stack()

def human_move(game_state):
    p = prompt("Which pile will you remove from?")
    n = prompt("How many coins do you want to remove?")
    if int(p) == 0:
        return handle_undo(game_state)
    else:
        push(game_stack,game_state)
        return remove_coins_from_pile(game_state,
                                      int(n), int(p))
```

Changes to Nim

```
def handle_undo(game_state):
    old_state = pop(game_stack)
    if old_state:
        display_game_state(old_state)
        return human_move(old_state)
    else:
        print("No more previous moves!")
        return human_move(game_state)
```


2048
SCORE
6380
BEST
7176
Join the numbers and get to the **2048 tile!**
New Game
2
4
16
4
8
16
64
128
2
128
512
2
2
4
32
64

Data Structures: Design Principles

When designing a data structure, need to spell out:

- Specification
- Implementation

Specification (contract)

- What does it do?
- Allows others to use it.

Implementation

- How is it realized?
- Users do not need to know this.
- Choice of implementation.

<u>Nim: Game state</u>

piles, coins in each pile

size, remove-coin

Multiple representations possible

Specification

- Conceptual description of data structure.
  - Overview of data structure.
  - State assumptions, contracts, guarantees.
  - Give examples.

Specification

- Operations:
  - Constructors
  - Selectors (Accessors)
  - Predicates
  - Printers

Example: Lists

- Specs:
  - A list is a collection of objects, in a given order.
    - e.g. [], [3, 4, 1]

Example: Lists

Specs:

- Constructors: `list(), [ ]`
- Selectors: `[ ]`
- Predicates: `type, in`
- Printer: `print`

Sets: Specs

A set is an unordered collection of objects (numbers, in our example) without duplicates.

- $\{3, 1, 2\}$, $\{1, 2, 3\}$ are the same
- `empty_set` is empty set
- $\{3, 3, 1, 2\}$ is not valid!

Sets: Specs

Constructors:

`make_set`, `adjoin_set`,
`union_set`, `intersection_set`

Selectors:

Predicates:

`is_element_of_set`, `is_empty_set`

Printers:

`print_set`

Sets: Contract

For any set S, and any object x

```
>>> is_element_of_set(x, adjoin_set(x, S))
True
```

Adjoining an object to a set produces a set that contains the object.

Sets: Contract

```
>>> is_element_of_set(x, union_set(S, T))
```

is equal to

```
>>> is_element_of_set(x, S) or
    is_element_of_set(x, T)
```

The elements of $(S \cup T)$ are the elements that are in S or in T

```
>>> is_element_of_set(x, empty_set)
False
```

No object is an element of the empty set.

etc…

Implementation

Choose a representation

- Usually there are choices, e.g. lists, trees
- Different choices affect time/space complexity.
- There may be certain constraints on the representation. These should explicitly stated.
    - e.g. in rational number package, denom ≠ 0

Implementation

- Implement constructors, selectors, predicates, printers, using your selected representation.
- Make sure you satisfy all contracts stated in specification!

Sets: Implementation #1

- Representation: unordered list
  - Empty set represented by empty list.
  - Set represented by a list of the objects.
  - Must take care to avoid duplicates

Sets: Implementation #1

Constructors:

```
def make_set():
    '''returns a new, empty set'''
    return []
```

Predicates:

```
def is_empty_set(s):
    return not s
```

Sets: Implementation #1

Predicates:

```
def is_element_of_set(x, s):
    if is_empty_set(s):
        return False
    for e in s:
        if e == x:
            return True
    return False
```

Time complexity:

$O(n)$, n is size of set

Sets: Implementation #1

Constructors:

```
def adjoin_set(x, s):
    if not is_element_of_set(x, s):
        s.append(x) # don’t add if already in
    return s
```

Time complexity:

$O(n)$

Sets: Implementation #1

Constructors:

```
def intersection_set(s1, s2):
        <...........>
        return <.....>
```

left as an exercise

(Question of the day!)

Sets: Implementation #2

- Representation: ordered list
  - Empty set represented by empty list.
  - Must take care to avoid duplicates.
  - But now objects are sorted.

WHY WOULD WE WANT TO DO THIS?

Sets: Implementation #2

Note: specs does not require this, but we can impose additional constraints in implementation.

But this is only possible if the objects are comparable, i.e. concept of “greater_than” or “less_than”.

e.g. numbers: <

e.g. strings, symbols: lexicographical order (alphabetical)

Not colors: red, blue, green

Sets: Implementation #2

Constructors:

```
def make_set():
        return []  #as before
```

Predicates:

```
def is_empty_set(s):
    return not s #as before
```

Sets: Implementation #2

Predicates:

```
def is_element_of_set(x, s):
    if is_empty_set(s):
        return False
    for e in s:
        if e == x:
            return True
        elif e > x:
            return False
    return False
```

Time complexity:

$O(n)$, *but faster than previous!*

Intersection

```
Set 1: (1  3  4  8)
Set 2: (1  4  5  6  8  9)
Result: ()
```

→ so 1 in intersection, move set1, set2 cursor forward

```
Set 1: (1  3  4  8)
Set 2: (1  4  5  6  8  9)
Result: (1)
```

→ 3 < 4, 3 not in intersection, forward set1 cursor only

Intersection

```
Set 1: (1  3  4  8)
Set 2: (1  4  5  6  8  9)
Result: (1)
```

→ so 4 in intersection, forward set1, set2 cursor

```
Set 1: (1  3  4  8)
Set 2: (1  4  5  6  8  9)
Result: (1  4)
```

→ 8 > 5, 5 not in intersection, forward set2 cursor only

Intersection

```
Set 1: (1  3  4  8)
Set 2: (1  4  5  6  8  9)
Result: (1  4)
```

→ 8 > 6, 6 not in intersection, forward set2 cursor

```
Set 1: (1  3  4  8)
Set 2: (1  4  5  6  8  9)
Result: (1  4)
```

→ so 8 in intersection, forward set1, set2 cursor

Intersection

```
Set 1: (1  3  4  8)
Set 2: (1  4  5  6  8  9)
Result: (1  4  8)
```

→set1 empty, return result

Sets: Implementation #2

Constructors:

```
def intersection_set(s1, s2):
    result = []
    i, j = 0, 0
    while i < len(s1) and j < len(s2):
        if s1[i] == s2[j]:
            result.append(s1[i])
            i += 1
            j += 1
        elif x1 < x2:
            i += 1
        else:
            j += 1
    return result
```

Time complexity:
$O(n)$, faster than previous!

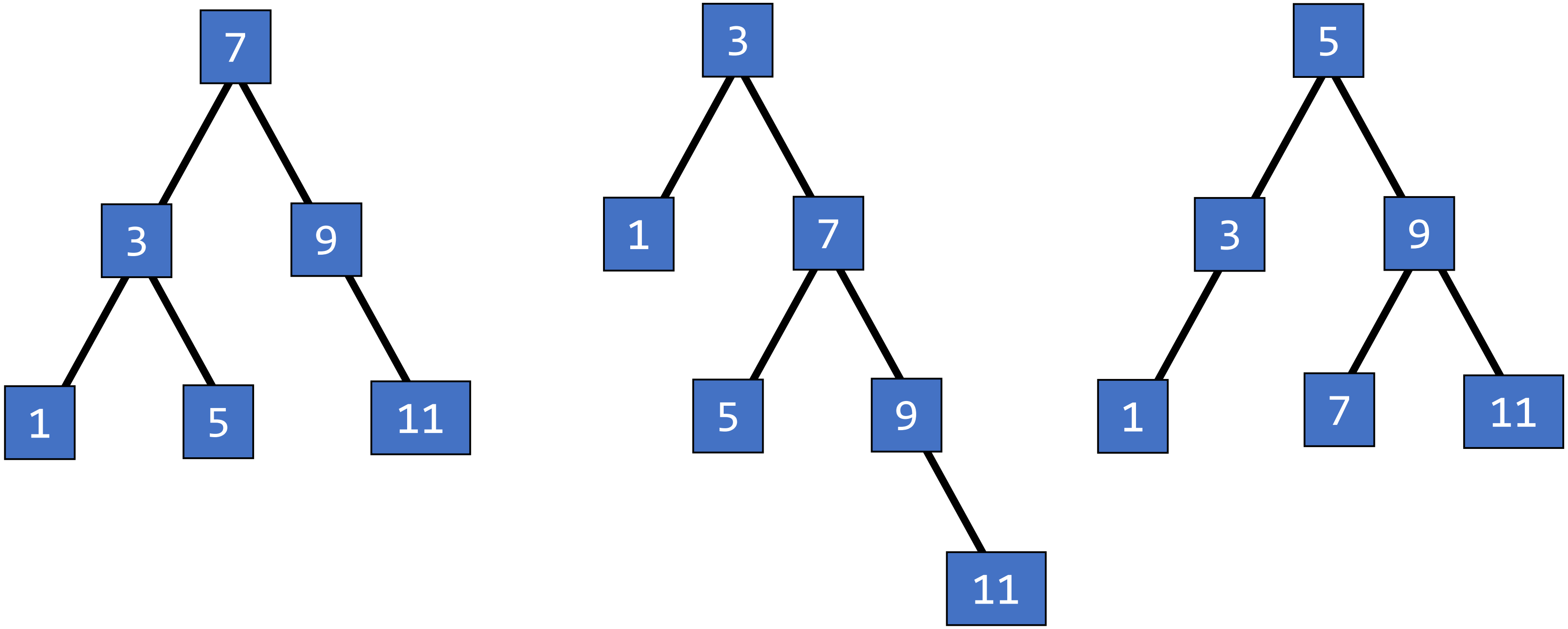
Sets: Implementation #3

- Representation: binary tree
  - Empty set represented by empty tree.
  - Must take care to avoid duplicates.
  - Objects are sorted.

Sets: Implementation #3

- Each node stores 1 object.
- Left subtree contains objects smaller than this.
- Right subtree contains objects greater than this.

Sets: Implementation #3



Three trees representing the set {1,3,5,7,9,11}

Sets: Implementation #3

Tree operators:

```
def make_tree(entry, left, right):
      return (entry, left, right)

def entry(tree):
      return tree[0]

def left_branch(tree):
        return tree[1]

def right_branch(tree):
        return tree[2]
```

Sets: Implementation #3

Predicates:

```python
def is_element_of_set(x, s):
    if is_empty_set(s):
        return False
    elif x == entry(s):
        return True
    elif x < entry(s):
        return is_element_of_set(x, left_branch(s))
    else:
        return is_element_of_set(x, right_branch(s))
```

Time complexity:

$O(\log n)$

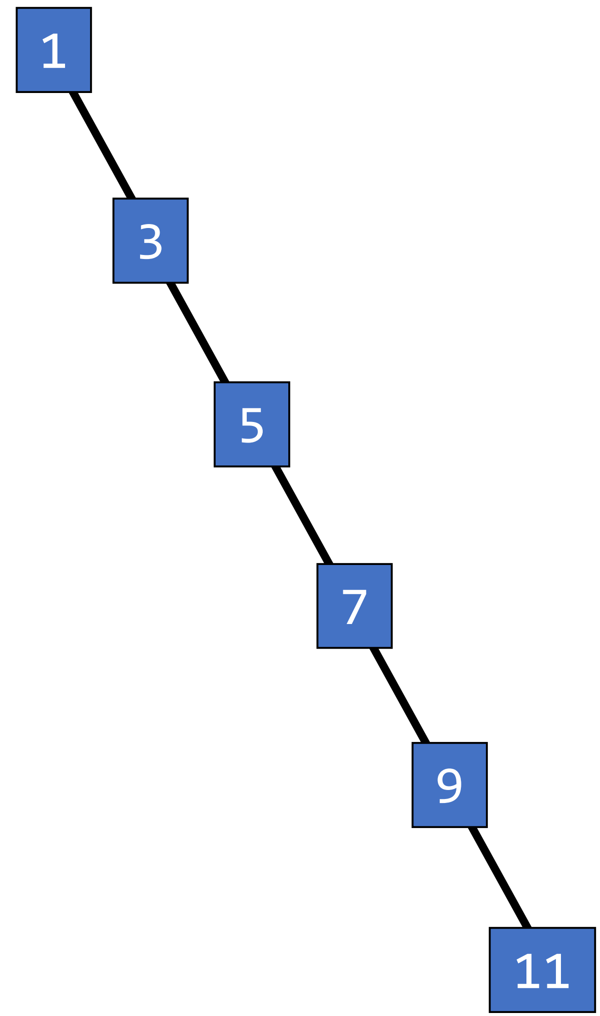
Sets: Implementation #3

Adjoin:

```
def adjoin_set(x, s):
    if is_empty_set(s):
        return make_tree(x, [], [])
    elif x == entry(s):
        return s
    elif x < entry(s):
        return make_tree(entry(s),
                         adjoin_set(x, left_branch(s)),
                         right_branch(s))
    else:
        return make_tree(entry(s), left_branch(s),
                         adjoin_set(x, right_branch(s))
```

Balancing trees

- Operation is $O(\log n)$ assuming that tree is balanced.
- But they can become unbalanced after several operations.
    - Unbalanced trees break the log n complexity.
- One solution: define a function to restore balance. Call it every so often.



Question of the Day

- How do we convert an unbalanced binary tree into a balanced tree?
- Write a function `balance_tree` that will take a binary tree and return a balanced tree (or as balanced as you can make it)

Multiple representations

- You have seen that for compound data, multiple representations are possible:
  - e.g. sets as:
    1. Unordered lists, w/o duplicates
    2. Ordered lists, w/o duplicates
    3. Binary trees, w/o duplicates

Multiple representations

- Each representation has its pros/cons:
  - Typically, some operations are more efficient, some are less efficient.
  - “Best” representation may depend on how the object is used.

Typically in large software projects, multiple representations co-exist.

Why?

Many possible reasons

- Because large projects have long lifetime, and project requirements change over time.
- Because no single representation is suitable for every purpose.
- Because programmers work independently and develop their own representations for the same thing.

Multiple representations

Therefore, you must learn to manage different co-existing representations.

- What are the issues?
- What strategies are available?
- What are the pros/cons?

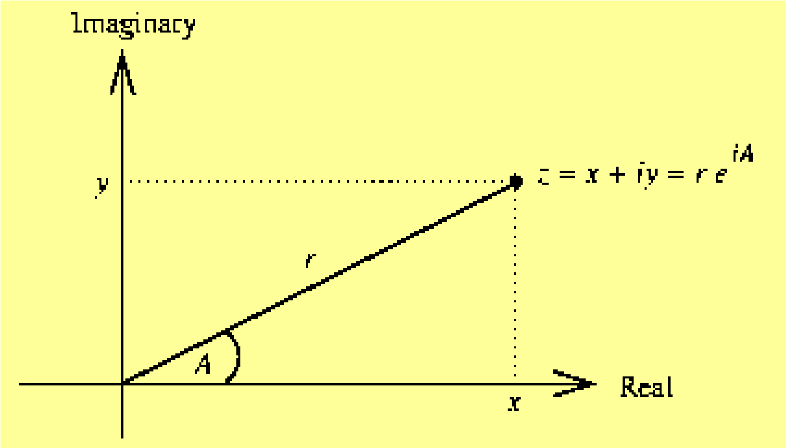
Complex-arithmetic package

- Recall: complex numbers

  $3 + 4i, i = \sqrt{-1}$

  $x + iy$, x: real part, y: imaginary part

- The above is rectangular form.
- There is also polar form:
  - $z = x + iy = re^{-iA}$
  - r: magnitude
  - A: angle



Abstraction barrier

Programs that use complex numbers

You want to build this

`add_complex, sub_complex, mul_complex, div_complex`

Complex Numbers Package

Rectangular representation	Polar representation

Wai Kay has provided this

Karen has provided this

Arithmetic package

Similar to rational number package (Lecture 7)

Addition: `z = z1 + z2`

```
real_part(z) = real_part(z1) +
               real_part(z2)
imag_part(z) = imag_part(z1) +
               imag_part(z2)
```

Arithmetic package

Multiplication: `z = z1 * z2`

```
magnitude(z) = magnitude(z1) *
               magnitude(z2)
angle(z) = angle(z1) + angle(z2)
```

etc.

These are easily implemented assuming the existence of

selectors: `real_part, imag_part, magnitude, angle`

constructors: `make_from_real_imag, make-from-mag-ang`

Wai Kay's (rectangular) code

```
def make_from_real_imag(x, y):
    return (x, y)
def real_part(z):
    return z[0]
def imag_part(z):
    return z[1]

def magnitude(z):
    return math.hypot(imag_part(z),
                      real_part(z))
def angle(z):
    return math.atan(imag_part(z) / real_part(z))

def make_from_mag_ang(r, a):
    return (r * math.cos(a), r * math.sin(a))
```

Karen's (polar) code

```
def make_from_mag_ang(r, a) :
    return (r, a)
def real_part(z):
    return magnitude(z) * math.cos(angle (z))
def imag_part(z):
    return magnitude(z) * math.sin(angle(z))

def magnitude(z):
    return z[0]
def angle(z):
    return z[1]
def make_from_real_imag(x, y):
    return (math.hypot(x, y), math.atan(y / x))
```

Python math library

```
import math

hypot(x, y) # Returns the square root of x2 + y2
sin(a)
cos(a)
atan(a)
atan2(y, x) # equivalent to atan(y/x)
            # in radians
```

Whose code is better?

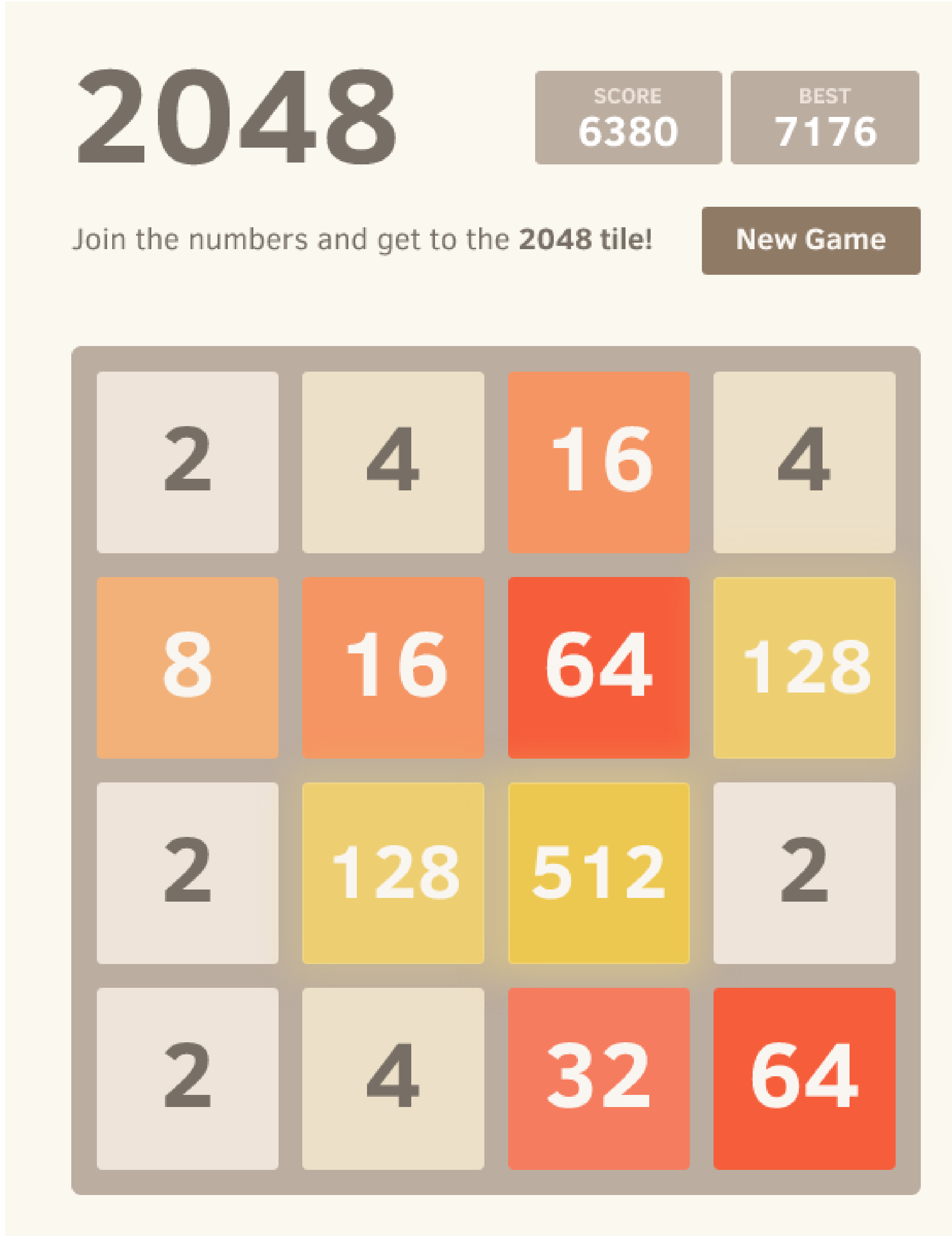
It depends

Summary

- Lots of wishful thinking (top-down)
- Design Principles
  - Specification
  - Implementation
- Abstraction Barriers allow for multiple implementations
- Choice of implementation affects performance!

If you have a lot of time on your hands….

- Play nim (dumb version)
- Re-write nim to allow for arbitrary number of piles of coins
- Write a smarter version of `computer-move`


2048
SCORE
6380
BEST
7176
Join the numbers and get to the **2048 tile!**
New Game
2
4
16
4
8
16
64
128
2
128
512
2
2
4
32
64